Clase 7 — 16.12.25

#IntelliJIDEA #JAVA #XML #DOM #SAX #JAXB #JSON

👨‍🏫 **Profesor:** Álvaro García Gutiérrez
📘 **Acceso a Datos**
🗓️ **Clase 7 — 16/12/2025**
🎯 **Tema: Proyecto Maven en VS Code + configuración (JAVA_HOME) + conversión XML → JSON (Jackson) + repaso JAXB**

1️⃣ ¿Qué es Maven y por qué se usa?

Maven es un **gestor de proyectos y dependencias** para Java. No es solo una herramienta para "bajar librerías", sino una **forma estandarizada de construir, organizar y ejecutar proyectos Java** de manera consistente en cualquier equipo.

La idea central de Maven es muy simple pero muy potente:
👉 *si alguien tiene tu proyecto y tu `pom.xml`, puede reconstruirlo exactamente igual en otro ordenador*.

Esto es lo que se conoce como un proyecto **portable y reproducible**.

🧩 ¿Qué problemas reales resuelve Maven?

Antes de Maven (o sin usarlo correctamente), lo habitual era:

- Tener carpetas llenas de `.jar` copiados a mano
- Versiones distintas de la misma librería según el ordenador
- Proyectos que funcionaban en un PC… y en otro no
- Builds manuales difíciles de repetir

Maven elimina todo eso porque:

- ✅ **Descarga automáticamente las librerías necesarias**, y siempre en la versión exacta indicada.
- ✅ **Define una estructura estándar** que todos los proyectos Maven comparten.
- ✅ **Automatiza tareas habituales**: compilar, ejecutar tests, empaquetar en `.jar`.
- ✅ **Hace explícitas las dependencias**: no hay magia oculta.

Por eso se dice (con cariño) que Maven evita el clásico:

> "En mi PC funciona"
> *(sí, pero en el del profe no 😅)*

📄 El papel central del `pom.xml`

El corazón de cualquier proyecto Maven es el archivo **`pom.xml`**
(Project Object Model).

Este archivo **no es opcional** ni un simple formulario: es el **contrato del proyecto**.

En él se define:

- 🔹 **Identidad del proyecto**
  - `groupId` → quién eres (normalmente dominio invertido)

    - `artifactId` → nombre del proyecto
    - `version` → versión actual del artefacto
- 🔹 **Versión de Java**
    - Qué versión se usa para compilar
    - Qué versión se espera en ejecución
- 🔹 **Dependencias**
    - Librerías externas como **Jackson**, **JAXB**, **JUnit**, etc.
    - Maven se encarga de descargarlas y mantenerlas actualizadas
- 🔹 **Plugins**
    - Cómo se compila
    - Cómo se empaqueta
    - Cómo se ejecutan tests
    - Cómo se generan artefactos finales

📌 **Conclusión importante de clase**
Por eso el profesor insiste en que *"lo primero es editar el `pom.xml`"*.
Si el `pom.xml` está mal, **todo lo demás falla**, aunque el código sea correcto.

2️⃣ Extensiones necesarias en Visual Studio Code

Para trabajar con Maven en VS Code **no basta con tener Java instalado**.
VS Code necesita extensiones que le permitan **entender proyectos Java/Maven**.

Las extensiones clave son:

- **Extension Pack for Java**
- **Maven for Java** (si no viene incluida)

Estas extensiones permiten a VS Code:

- Detectar proyectos Maven automáticamente
- Interpretar el `pom.xml`
- Resolver dependencias
- Ofrecer acciones como *Build*, *Run*, *Test*
- Detectar errores de configuración (Java, Maven, PATH…)

☕ Java Runtime: detalle crítico (y fuente de errores)

Además de las extensiones, es imprescindible que:

- Tengas un **JDK instalado** (no solo JRE)
- VS Code **sepa dónde está ese JDK**

Por eso aparece en clase la opción:

- **Java: Configure Java Runtime**

📌 Si el JDK no está bien detectado:

- Maven puede no arrancar
- El proyecto puede no compilar

- Aparecen errores confusos (versiones, `-165`, etc.)

Esto explica por qué **a veces cerrar y abrir VS Code "arregla" cosas**:
VS Code vuelve a leer las variables de entorno del sistema.

3️⃣ Crear un proyecto Maven en VS Code

Una vez el entorno está listo, se crea el proyecto Maven desde VS Code.

🔹 3.1 Crear el proyecto

El flujo real que se sigue (tal como se ve en las imágenes) es:

1. Abrir la **Command Palette** con
   `Ctrl + Shift + P`
2. Escribir y seleccionar:
   Maven: Create Maven Project
3. Elegir el arquetipo:
   `maven-archetype-quickstart`

   📌 Este arquetipo crea:
   - Un proyecto Java básico
   - Con `main` y `test`
   - Estructura estándar Maven
4. Introducir los datos del proyecto:
   - `groupId`: por ejemplo `com.ejemplo`
   - `artifactId`: por ejemplo `xml-json-ejercicio-online`

Estos valores **no son decorativos**:

- Maven los usa para identificar el proyecto
- Determinan el nombre del paquete base
- Se reflejan en la estructura de carpetas

📂 Estructura generada por Maven

Tras la creación, VS Code genera automáticamente una estructura como:

- `src/main/java/`
- `src/test/java/`
- `pom.xml`

Esta estructura **no es arbitraria**:

- `src/main/java`
  → código principal de la aplicación
- `src/test/java`
  → tests (JUnit, etc.)
- `pom.xml`
  → configuración completa del proyecto

📌 Más adelante se añadirá también:

- `src/main/resources`
  para XML, JSON y otros ficheros no Java

🧠 Idea clave para entender Maven (muy de examen)

Maven **no es una librería**, es una **forma de trabajar**:

- El código Java vive en `src/main/java`
- Los recursos viven en `src/main/resources`
- Las dependencias **no se copian**, se declaran
- El proyecto se reconstruye desde el `pom.xml`

Si entiendes esto, entiendes el 80 % de Maven.

4️⃣ Ventajas del `pom.xml` (y por qué "lo primero es editarlo")

Nada más crear el proyecto Maven, el profesor insiste en **abrir y revisar el `pom.xml`**.
Esto no es un capricho: **todo el comportamiento del proyecto depende de este archivo**.

El `pom.xml` actúa como:

- 📜 Documento de identidad del proyecto
- ⚙️ Archivo de configuración de compilación
- 📦 Gestor de librerías externas
- 🧠 Fuente única de verdad del build

Si el `pom.xml` está mal, **el proyecto puede compilar a medias, fallar en ejecución o ni siquiera arrancar**, aunque el código Java esté correcto.

🔹 4.1 Versión de Java (propiedades)

Uno de los primeros bloques que se revisa o se añade es el de **propiedades**, donde se fija explícitamente la versión de Java.

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>17</maven.compiler.source>
  <maven.compiler.target>17</maven.compiler.target>
</properties>
```

Aquí se están definiendo **tres cosas clave**:

- 🧾 **Codificación del proyecto** (`UTF-8`)
  Evita problemas con acentos, eñes y caracteres especiales en XML, JSON o strings.
- ☕ **Versión de Java de compilación** (`source`)
  Indica qué sintaxis de Java se permite usar.
- 🎯 **Versión objetivo de Java** (`target`)
  Indica para qué versión se genera el bytecode.

📌 **Por qué esto es tan importante en clase**
En varias capturas se ve que Maven crea proyectos con valores antiguos (1.7 / 1.8).
Eso provoca situaciones como:

- Errores de compilación sin sentido aparente
- Librerías modernas (Jackson, JAXB nuevo) que no funcionan
- Avisos constantes en consola
- Código que compila en IntelliJ pero no en VS Code (o al revés)

👉 **Conclusión de examen**:
Si Java y Maven no están alineados en el `pom.xml`, el proyecto es inestable.

🔹 4.2 Dependencias (para XML → JSON)

Para la conversión de **XML a JSON**, en clase se utiliza la librería **Jackson**, que se añade como dependencia Maven.

```
<dependencies>
  <!-- Jackson core / JSON -->
  <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.16.1</version>
  </dependency>

  <!-- Jackson XML -->
  <dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
    <version>2.16.1</version>
  </dependency>
</dependencies>
```

Aquí hay una idea clave que conviene interiorizar:

- **No se copia ningún `.jar` al proyecto**
- Solo se **declara qué se necesita**
- Maven se encarga de:
  - Descargarlo
  - Guardarlo en el repositorio local
  - Resolver dependencias internas

📌 **Qué aporta cada dependencia**:

- `jackson-databind`
  → Permite trabajar con JSON, convertir objetos, árboles (`JsonNode`), serializar/deserializar.
- `jackson-dataformat-xml`
  → Añade la capacidad de leer XML usando la misma filosofía de Jackson.

💡 Gracias a esto, **XML y JSON se tratan casi igual** desde el punto de vista del código.

🧠 Idea clave (muy importante)

Jackson **no convierte directamente XML → JSON como texto plano**, sino que:

1. Lee el XML

2. Lo convierte en una **estructura de árbol** (`JsonNode`)
3. Ese árbol se vuelve a serializar como JSON

Esto es exactamente lo que se explica luego en el código.

5️⃣ Crear `resources` y un XML de prueba

Una vez el proyecto y el `pom.xml` están preparados, el siguiente paso es **crear recursos de entrada**.

Dentro de `src/main/` se crea la carpeta:

📁 `resources`

Y dentro se añade un XML de prueba, como por ejemplo `yo.xml` :

```xml
<?xml version="1.0" encoding="UTF-8"?>
<yo>
  <nombre>Tu nombre</nombre>
  <edad>Tu edad</edad>
</yo>
```

📌 Este XML no es especial por su contenido, sino por **dónde está colocado**.

🔍 ¿Por qué `resources` y no cualquier carpeta?

Porque Maven trata `src/main/resources` de forma especial:

- Todo lo que hay ahí:
  - Se copia automáticamente al build
  - Se incluye en el **classpath**
- Es accesible desde Java **sin usar rutas absolutas**

Esto permite escribir código portable, sin rutas dependientes del sistema operativo.

📌 Consecuencia directa (muy importante)

Gracias a esto, el XML puede cargarse así:

```java
getClassLoader().getResourceAsStream("yo.xml");
```

Y **funcionará igual**:

- En Windows
- En Linux
- En otro ordenador
- En el PC del profesor

6️⃣ Conversión XML → JSON (código base)

El ejemplo de clase utiliza una clase Java dedicada exclusivamente a la conversión, por ejemplo `XmlToJson.java` .

El código sigue un flujo muy claro y didáctico.

🧩 Flujo real del programa

1️⃣ Carga del XML desde `resources`

```
InputStream is = XmlToJson.class
    .getClassLoader()
    .getResourceAsStream("yo.xml");
```

Aquí ocurren varias cosas importantes:

- Se accede al **classpath**
- No se usa una ruta física (`C:\...`)
- Si el archivo no existe, `is` será `null`

Por eso se comprueba inmediatamente:

```
if (is == null) {
    System.out.println("No se encontró yo.xml en resources");
    return;
}
```

2️⃣ Conversión del XML a texto

```
String xml = new String(is.readAllBytes(), StandardCharsets.UTF_8);
```

Se lee todo el contenido del XML y se guarda como `String` , respetando UTF-8.

3️⃣ XML → estructura de datos (árbol)

```
XmlMapper xmlMapper = new XmlMapper();
JsonNode node = xmlMapper.readTree(xml.getBytes(StandardCharsets.UTF_8));
```

Aquí ocurre la magia real:

- `XmlMapper` entiende la sintaxis XML
- `readTree()` convierte el contenido en un **árbol genérico**
- Ese árbol ya **no es XML**, es una estructura interna tipo JSON

4️⃣ Árbol → JSON “bonito”

```
ObjectMapper jsonMapper = new ObjectMapper();
String json = jsonMapper
    .writerWithDefaultPrettyPrinter()
    .writeValueAsString(node);
```

Esto genera un JSON:

- Legible
- Indentado

- Fácil de depurar y entender

🔹 Puntos clave del enfoque

- `XmlMapper` → interpreta XML
- `readTree()` → crea una estructura intermedia
- `ObjectMapper` → genera el JSON final

📌 **Ventaja del enfoque**:
No dependes de clases Java específicas (POJOs).
Funciona con **cualquier XML**, aunque no conozcas su estructura a priori.

7️⃣ Posibles errores al crear/usar Maven (los que salieron en clase)

Esta parte es especialmente importante porque **los errores son reales**, no teóricos.

❌ 7.1 `mvn` no se reconoce (terminal)

Al ejecutar:

```
mvn -v
```

El sistema responde que no reconoce el comando.

Esto indica casi siempre uno de estos casos:

- Maven no está instalado
- Maven está instalado pero **no está en el PATH**
- VS Code se abrió **antes** de configurar Maven

Por eso en clase se hace:

- Cerrar VS Code
- Volver a abrirlo
- Probar otra vez

📌 **Idea clave**:
VS Code hereda las variables de entorno **al arrancar**, no en caliente.

❌ 7.2 Error `-165` y `JAVA_HOME`

El error `-165`, junto con referencias a `JAVA_HOME`, apunta a un problema muy concreto:

👉 **El JDK no está bien configurado o no es detectable por las herramientas Java**

Esto suele ocurrir cuando:

- Solo hay JRE
- Hay varios JDK instalados
- `JAVA_HOME` apunta a una ruta incorrecta

✅ Solución aplicada en clase

1. Instalar un **JDK completo**
2. Definir correctamente:
   - `JAVA_HOME`
3. Añadir:
   - `%JAVA_HOME%\bin` al PATH
4. En VS Code:
   - `Java: Configure Java Runtime`
   - Seleccionar el JDK correcto
5. **Cerrar y abrir VS Code**

📌 Síntoma típico:
Todo "parece" instalado, pero hasta reiniciar VS Code **no funciona**.

8️⃣ Mini repaso: JAXB (lo que aparece en las capturas)

JAXB (Java Architecture for XML Binding) es una API estándar de Java cuyo objetivo es **mapear XML ↔ objetos Java** de forma automática.
Aquí no se trabaja con nodos genéricos (como en DOM/SAX o Jackson), sino con **clases Java concretas** que representan la estructura del XML.

La idea clave es el **data binding**:
👉 enlazar etiquetas XML con atributos de una clase Java mediante anotaciones.

🔁 Operaciones fundamentales en JAXB

En las capturas de IntelliJ se ven claramente las dos operaciones básicas:

- **Unmarshal** → XML → Objeto Java
- **Marshal** → Objeto Java → XML

Estas dos operaciones forman un ciclo completo de trabajo con XML.

🔹 Unmarshal (XML → Objeto Java)

Durante el *unmarshal*:

1. JAXB lee el XML desde un fichero (`coche.xml`)
2. Analiza su estructura
3. Crea una instancia de la clase Java correspondiente (`Coche`)
4. Rellena automáticamente sus atributos

Ejemplo conceptual (no literal):

```
Unmarshaller um = ctx.createUnmarshaller();
Coche coche = (Coche) um.unmarshal(new File("coche.xml"));
```

📌 En este punto:

- El XML **desaparece**
- Trabajas únicamente con un **objeto Java real**
- Puedes usar getters, setters, lógica de negocio, etc.

Esto se ve en las imágenes cuando:

- Se imprime la marca y el modelo
- Se modifica el modelo (`Ibiza → León`)

🔹 Marshal (Objeto Java → XML)

Tras modificar el objeto en memoria, JAXB permite **volver a generar XML**.

```
Marshaller m = ctx.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
m.marshal(coche, System.out);
```

Aquí ocurre lo contrario:

- El objeto Java se serializa
- Se genera un XML nuevo
- Puede mostrarse por consola o guardarse en un fichero

📌 En las capturas se ve:

- `coche_modificado.xml`
- Con el nuevo valor del campo `modelo`
- XML bien formateado (`JAXB_FORMATTED_OUTPUT`)

🧠 Concepto clave: JAXB ≠ Jackson

Es importante no mezclar conceptos:

- **JAXB**
  - Trabaja con **clases Java**
  - Necesita que la estructura XML coincida con la clase
  - Ideal cuando el XML tiene un esquema claro y fijo
- **Jackson (XML → JSON)**
  - Trabaja con **estructuras genéricas**
  - No necesita clases específicas
  - Ideal para transformar formatos o trabajar con XML desconocido

Ambos enfoques **no se excluyen**, se usan según el problema.

⚠️ Validación del XML (detalle importante de teoría)

En la teoría aparece una advertencia muy clara:

> JAXB no trabaja bien si el XML no es válido.

Esto significa que:

- Si el XML está mal formado
- Si faltan etiquetas obligatorias
- Si la estructura no coincide con la clase Java

👉 JAXB lanzará excepciones y **no creará el objeto**.

Por eso, en proyectos reales:

- Se valida el XML antes (DTD / XSD)
- O se controla muy bien su origen

9️⃣ Checklist rápido (para que te funcione a la primera)

Este bloque resume **todo lo que en clase puede fallar** y sirve como guía de depuración.

✅ Entorno

Este checklist corresponde al **sistema**, no al proyecto:

- ☑️ **JDK instalado** (por ejemplo Java 17)
- ☑️ Variable `JAVA_HOME` correctamente definida
- ☑️ `%JAVA_HOME%\bin` incluido en el `PATH`
- ☑️ `java -version` devuelve la versión esperada
- ☑️ Maven instalado (o Maven Wrapper del proyecto)
- ☑️ `mvn -v` funciona desde PowerShell / terminal

📌 Si cualquiera de estos puntos falla:

- Maven puede no arrancar
- VS Code puede no detectar Java
- Aparecen errores “raros” (como el `-165`)

✅ Proyecto

Este checklist corresponde al **proyecto Maven**:

- ☑️ `pom.xml` con versión de Java correcta
- ☑️ Dependencias Jackson correctamente añadidas
- ☑️ Carpeta `src/main/resources` creada
- ☑️ XML (`yo.xml`, `coches.xml`, etc.) dentro de `resources`
- ☑️ Clase `XmlToJson` compila sin errores
- ☑️ El XML se encuentra vía `getClassLoader()`

📌 Si el XML no se encuentra:

- Normalmente **no está en** `resources`
- O el nombre no coincide exactamente

1️⃣0️⃣ Instalación de Maven en Windows (paso a paso)

> 🎯 Objetivo real de este apartado:
> que `mvn` funcione **fuera** y **dentro** de VS Code.

Este proceso es exactamente el que se ve reflejado en las capturas y en la explicación del profesor.

🔹 1.1 Descargar Maven (binario)

Se accede a la página oficial de **Apache Maven** y se descargan los archivos disponibles.

Es fundamental elegir correctamente:

- ✅ **ZIP binario (`*-bin.zip`)** → instalación
- ❌ **ZIP/TAR "src"** → código fuente (no sirve para ejecutar Maven)

Ejemplo correcto:

- `apache-maven-3.9.11-bin.zip`

📌 Error típico de alumnado:
descargar el `src.zip` y preguntarse por qué no existe `mvn.cmd`.

🔹 1.2 Extraer Maven en una ruta "limpia"

Una vez descargado:

1. Botón derecho → **Extraer todo**
2. Elegir una ruta sencilla, por ejemplo:
   - `C:\dev\apache-maven-3.9.11\`
   - `C:\apache-maven-3.9.11\`

⚠️ Se insiste en evitar:

- Rutas con espacios
- Carpetas protegidas del sistema

Esto evita problemas con permisos y PATH.

🔹 1.3 Configurar variables de entorno (PATH)

Este es **el paso crítico** que en clase provoca el error de `mvn`.

Pasos reales:

1. Buscar en Windows:
   - **Editar las variables de entorno del sistema**
2. Abrir **Variables de entorno**
3. En *Variables del sistema*:
   - Seleccionar `Path`
   - Pulsar **Editar**
4. Añadir una nueva entrada:
   - `C:\dev\apache-maven-3.9.11\bin`
5. Aceptar todas las ventanas

🔧 Variable opcional (pero profesional)

Aunque no es estrictamente necesaria, es correcto añadir:

- `MAVEN_HOME = C:\dev\apache-maven-3.9.11`

Ayuda a:

- Claridad del entorno
- Compatibilidad con otras herramientas

📌 **Detalle clave que se ve en clase**
Después de configurar Maven:

- Se **cierra VS Code**
- Se vuelve a abrir
- Se prueba `mvn -v`
- **Ahora sí funciona**

Esto no es magia: VS Code **no recarga el PATH en caliente**.

1️⃣1️⃣ Verificación en PowerShell (antes de VS Code)

Una vez configurado Maven y las variables de entorno, **el primer sitio donde hay que comprobar que todo funciona es fuera de VS Code**, es decir, en una terminal del sistema (PowerShell o CMD).

📌 **Detalle clave que se recalca en clase**
Las variables de entorno **no se actualizan en terminales ya abiertas**.
Por eso es obligatorio **abrir una terminal nueva** tras modificar el `PATH`.

🔍 Comprobación básica

En una **nueva** ventana de PowerShell:

```
mvn -v
```

Si Maven está correctamente instalado, la salida muestra:

- 📦 **Versión de Maven**
- ☕ **Java runtime detectado** (JDK en uso)
- 💻 **Sistema operativo**

Esto confirma tres cosas a la vez:

- Maven existe
- Está en el PATH
- Está enlazado con un JDK válido

🔹 Si sigue fallando

Si aparece el mensaje:

> “mvn no se reconoce como un comando interno o externo…”

Entonces el problema **no está en el proyecto**, sino en el sistema.
Las causas habituales (todas vistas en clase) son:

- ❌ La ruta `...\apache-maven\bin` **no está bien añadida** al PATH
- ❌ La terminal estaba abierta **antes** de cambiar las variables

- ❌ La ruta añadida **no es la carpeta** `bin`
  (debe ser la que contiene `mvn.cmd`)

📌 **Conclusión importante**
Hasta que `mvn -v` no funcione en PowerShell, **VS Code tampoco podrá usar Maven**.

1️⃣2️⃣ Reinicio de VS Code y comprobación

Este paso parece trivial, pero en la práctica **es decisivo**, y por eso el profesor lo hace explícitamente en clase.

🔁 Proceso seguido

1. **Cerrar completamente VS Code**
2. Volver a **abrir VS Code**
3. Abrir una **terminal integrada**
4. Ejecutar de nuevo:

```
mvn -v
```

✅ Resultado esperado

Ahora Maven **sí es reconocido** dentro de VS Code.

Esto ocurre porque:

- VS Code **lee las variables de entorno solo al arrancar**
- Al reiniciarlo, hereda el PATH actualizado

📌 Este paso explica por qué:

- Maven “está instalado”
- PowerShell funciona
- Pero VS Code no lo detecta hasta reiniciar

No es un bug: es funcionamiento normal del entorno.

1️⃣3️⃣ Ejecutar el proyecto: XML → JSON con Jackson

Con Maven ya operativo, se pasa al **objetivo real del ejercicio**:
convertir un XML a JSON usando **Jackson** dentro de un proyecto Maven.

🔹 13.1 Preparación mínima del proyecto

Para que el ejercicio funcione correctamente, deben cumplirse **tres condiciones**:

- 📄 **`pom.xml`**
  - Contiene las dependencias:
    - `jackson-databind`
    - `jackson-dataformat-xml`
- 📁 **`src/main/resources`**
  - Contiene el XML de entrada (`yo.xml`, `coches.xml`, etc.)

- ☕ **Clase Java principal**
  - Lee el XML desde `resources`
  - Lo convierte a `JsonNode`
  - Genera el JSON resultante

📌 Si falla algo aquí, el error **no es de Jackson**, sino de estructura del proyecto.

🔹 13.2 Ejecución típica del proyecto

Una vez todo está configurado, el flujo normal es:

```
mvn clean compile
```

Este comando:

- Limpia compilaciones anteriores
- Compila el proyecto
- Descarga dependencias si faltan

Después:

- Se ejecuta el `main` desde VS Code (botón ▶️)
- O mediante configuración de ejecución

✅ Resultado esperado

Si todo está correcto:

- 🖥️ **Consola**
  - Muestra el XML original
  - Muestra el JSON generado
- 📄 **Proyecto**
  - Aparece el fichero JSON de salida (por ejemplo `coches.json`)

📌 Detalle importante visto en clase
Si el nombre del fichero es solo `coches.json`, sin ruta:

- Se crea en la **raíz del proyecto**
- Es decir, junto al `pom.xml`

1️⃣4️⃣ Ejemplo mostrado en IntelliJ (JAXB)

En las capturas finales se muestra un **ejercicio equivalente**, pero realizado en **IntelliJ**, usando **JAXB** en lugar de Jackson.

El proyecto tiene una estructura tipo **`jaxb-demo`**, muy clara y didáctica.

📂 Estructura del proyecto JAXB

- `src/main/java/JAXB/`
  - `Coche.java`
    → clase modelo con anotaciones JAXB

    - `MainJaxb.java`
      → clase principal
- `src/main/resources/`
    - `coche.xml`
      → XML de entrada
    - `coche_modificado.xml`
      → XML generado tras modificar el objeto
    - `coches.xml`
      → ejemplo con varios coches
- (Opcional) salida JSON si se combina con Jackson

📌 Aquí ya no se trabaja con nodos genéricos, sino con **clases Java reales**.

1️⃣5️⃣ Funcionamiento de `MainJaxb.java` (detalle conceptual)

Este ejercicio demuestra el **ciclo completo JAXB**: leer → modificar → escribir.

🔁 Flujo completo del programa

1. **Crear el contexto JAXB**

   ```
   JAXBContext ctx = JAXBContext.newInstance(Coche.class);
   ```

   JAXB analiza la clase `Coche` y sus anotaciones.
2. **Crear el Unmarshaller**

   ```
   Unmarshaller um = ctx.createUnmarshaller();
   ```

   Preparado para convertir XML → objeto.
3. **Leer el XML y generar el objeto**

   ```
   Coche coche = (Coche) um.unmarshal(new File("coche.xml"));
   ```

   El XML se transforma en una instancia Java.
4. **Modificar el objeto**
    - Ejemplo: cambiar el modelo
      `Ibiza → León`
5. **Crear el Marshaller**

   ```
   Marshaller m = ctx.createMarshaller();
   ```
6. **Activar formato legible**

   ```
   m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
   ```
7. **Generar el nuevo XML**
    - Por consola
    - Y/o guardarlo como `coche_modificado.xml`

📌 Idea clave del ejercicio (muy importante)
JAXB permite trabajar con XML **como si fuera Java puro**:

- El XML entra → se convierte en objeto

- Se aplica lógica Java
- Se vuelve a generar XML

Esto contrasta con Jackson, donde:

- No hay clases específicas
- Se trabaja con estructuras genéricas
